Broadview
www.broadview.com.cn

Windows 内核情景分析

——采用开源代码ReactOS

（下册）

毛德操 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Windows 内核原理与分析

基于现代处理器和Windows 10系统

[illegible]

[illegible]

Windows 内核情景分析

——采用开源代码ReactOS

（下册）

毛德操 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书通过分析具有“开源 Windows”之称的 ReactOS 的源代码，介绍 Windows 内核各个方面的结构、功能、算法与具体实现。全书从“内存管理”、“进程”、“进程间通信”、“设备驱动”等 14 个方面进行分析介绍。本书的写作仿照作者广受欢迎的《Linux 内核源代码情景分析》一书，所有的分析都有 ReactOS 的源代码（以及部分由微软公开的源代码）作为依据，使读者能深入、具体地理解 Windows 内核的方方面面，也使读者的软件开发能力和水平得到提高。

本书可以供大学有关专业的高年级学生和研究生用作教学参考，也可供广大的软件工程师，特别是从事系统软件研发的工程师用于工作参考或进修教材。

图书在版编目（CIP）数据

Windows 内核情景分析：采用开源代码 ReactOS. 下册 / 毛德操著. —北京：电子工业出版社，2009.5
ISBN 978-7-121-08114-9

I. W… II. 毛… III.窗口软件，Windows—程序设计 IV. TP316.7

中国版本图书馆 CIP 数据核字（2009）第 005798 号

责任编辑：朱沭红
印　　刷：北京东光印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　　印张：93.25　　字数：2395 千字
印　　次：2009 年 5 月第 1 次印刷
印　　数：3000 册　　定价：190.00 元（上、下册）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

目　录

上　册

下　册

第9章　设备驱动

9.1　Windows的设备驱动框架

Windows内核管理层的部件之一是I/O管理模块，有时候也称为I/O子系统。I/O管理模块所管理的对象与活动纵向贯穿管理层、核心层乃至HAL层，所以称之为子系统其实也有道理。I/O管理的主体就是我们所说的设备驱动。很自然地，如果我们沿着纵向考察某项设备的驱动，则一般而言也会分成若干层次。操作系统的一个基本原理就是分层虚拟，即使一种设备的驱动程序全部都在同一个源程序文件中，只要不是特别简单的设备驱动，其设计必然会自觉或不自觉地体现分层虚拟的原理。所以，一项设备的驱动软件常常表现为若干驱动程序的“堆叠（Stack）”。这个堆叠的顶层在管理层中，底层则在HAL层中；愈往上，离具体设备的硬件愈远，就愈抽象，与其他设备的共性就愈多；愈往下，离具体设备的硬件愈近，就愈具体、愈体现出具体设备的个性。不过上层模块与下层模块之间不一定是一对一的关系，而可以是一对多的关系。以文件系统为例，这个堆叠的顶层（大致上）是文件系统，下层是文件系统所在的载体。但是这个载体可以是磁盘，也可以是光盘，还可以是“U盘”。如果是磁盘，则可以是连接在IDE接口上的固定硬盘，也可以是SCSI磁盘。所以，设备驱动逻辑意义上的系统结构其实是一种（倒置的）树状结构，所谓一个堆叠实际上相当于从根节点通往某个特定叶节点的一条路径。

另一方面，设备驱动也是内核中最需要加以动态扩充的部分。这是因为在编译生成系统内核时常常无法确切地知道使用中究竟需要哪一些设备。显然，最好的办法是将各种设备驱动的堆叠做成可以动态安装的程序模块，就像在用户空间可以动态加载DLL一样。Windows正是这么做的，文件扩展名为.sys的模块就是此类可动态装载的内核模块。注意“模块”这个词在不同的语境下有不同的意义。当我们谈论内核管理层中的I/O管理模块时，是指逻辑上相对自成一体的一个部分，也许称之为“板块”更贴切一些。而在谈论.sys模块的时候，则是说一块可动态装载的可执行映像。这种可动态装载的可执行映像可大可小，事实上win32k.sys就是这样一个模块。当然，其他模块就没有这么大了。在实践中，一般都根据具体的需要把一种特定设备的驱动程序堆叠实现成一个.sys模块；或者把一个堆叠中的一层或几层实现成一个.sys模块，实际使用时则由一个或数个.sys模块提供该种设备的驱动程序堆叠。

所以，设备驱动有两个问题，一个是分层的问题，一个是动态装载的问题。

但是程序的分层有概念和形式之分，概念上的分层只是程序员编程的方法问题，当然里面也体现着程序员的技艺和水平；而形式上的分层，则是系统为设备驱动程序的开发定下的模型（Model）和框架（Framework），一方面要求开发者按特定的、体现着程序分层的模型开发设备驱动程序，另一方面则又为符合这种模型的设备驱动程序提供基础设施的支持。打个比方，就好像一个机架，一方面它要求凡是要插入这个机架的模块在形状、尺寸等方面都符合某种规定；而另一方面，只要你符合这样的规定，则机架为你提供电源、通风、模块间通信等基础设施。

对于 Windows 的设备驱动模块，这个框架定义了：

- 设备驱动模块以何种形式提供有关的操作（体现为一个含有若干函数指针的数据结构），以及这些操作的范围（打开、关闭、读、写等）。
- 怎样启动由设备驱动模块提供的特定操作（将“操作码”等参数组装在一个标准格式的数据结构“I/O 请求包”即 IRP 中，以此数据结构作为形式上的调用参数，通过设备驱动框架为此提供的手段 IoCallDriver()调用设备驱动模块提供的相应函数指针）。
- 如果需要，一个设备驱动模块如何启动其（同一堆叠中的）下层模块的操作（仍通过 IoCallDriver()，并把“I/O 请求包”传到下一层）。
- 可以从设备驱动程序中调用内核的哪一些函数，访问内核的哪一些变量（Windows 的 DDK 对此做出了规定）。

之所以对于设备驱动模块的界面可以定义一种固定的模型，是因为设备驱动模块所提供的服务有个固定的范围，属于一个固定的集合，不外乎打开、关闭、读、写等操作。相比之下，用户空间的 DLL 就不像设备驱动模块那么规范，因为一般而言 DLL 所提供的服务五花八门，并不限于某个固定的集合，因而无法统一到一组固定的函数集合中。

不过，对于设备驱动的分层也不应采取教条的态度，有些设备的驱动确实很简单，因此实际上并不需要分层。比方说，外部设备是个继电器，对继电器的驱动只需要一对导线，只有“吸合”和“断开”两种状态，并且假定其接口板卡就插在 ISA 总线上，那么这样的设备驱动显然就不必分层了。依此类推，则“原始”的并行口、“原始”的 RS232 串行接口等的设备驱动也就不必分层。所以，Windows 早期的设备驱动是不分层的，因而称为“老式（Legacy）”设备驱动。注意所谓“老式”是指设备驱动而言，而不是指设备本身。

但是，对于比较复杂的外设，分层就显得必要了。例如网络设备，同一个具体链路层驱动（例如以太网驱动）的上面可能需要实现不同的网络层规程，没有理由要求网络层规程的实现者同时也来实现链路层规程，而且是与所有不同链路层驱动的组合。再如磁盘，磁盘是分区的，每个分区都是一个逻辑磁盘，最终又都映射到同一个物理磁盘上，而物理磁盘又可能是 IDE 磁盘或 SCSI 磁盘，这两种磁盘的驱动是不一样的。显然，这时候就需要分层实现了。

设备驱动分层的要求成为必需并且紧迫，则是由于“即插即用（Plug-and-Play）”即 PnP 设备的出现。PnP 设备要求内核密切注视总线和各种外设接口的状态变化，一有外设插入就向其询问有关

其设备类型、生产厂家等信息，然后根据这些信息（通过注册表查询）判定应该装载哪一些.sys 模块，并加以装载和初始化，可能还要启动相关的应用软件。对于这种实时的监测和装载，其影响的范围当然是愈小愈好，所以要把设备驱动的层次分得比较细才好。以 U 盘的插入为例，不同厂家的 U 盘可能在设备驱动方面有些特殊性，所以应该提供相应的.sys 模块。但是这些.sys 模块的内容应该是什么呢？整个文件系统的实现？显然不是，甚至也不需要触动 USB 设备这一层，因为所插入的设备同属 USB 设备，不同的只是某些细节，所以厂家提供的.sys 模块只相当于整个堆叠的底层，并且是很薄的一层。事实上，这样的驱动模块称为“小端口（Miniport）”驱动。

不过，并非所有的外设都是 PnP 设备，事实上直到现在 Windows XP 上还在使用着许多老式设备驱动。Windows 2000 的 DDK 提供了一个有用的工具 Devtree，用来观察内核中的“设备驱动树”；笔者的笔记本电脑用的是 Windows XP，但是用 Devtree 观察就看到内核中超过一半的设备驱动模块是“老式”模块，“老式”设备驱动是不堆叠的，但是也可以认为一个“老式”设备驱动就是一个堆叠。对于应用软件，“老式”设备都是“可见”的，即可以直接打开和操作的。有些“老式”设备的驱动在概念上已经有了分层的需要，但是这种分层只是体现在同一模块内部的程序结构上，而并没有从物理上将其划分实现到多个不同模块中，此种“单块”的设备驱动仍旧是“老式”的。

所以，在 Windows 的设备驱动框架中，最上层是内核的 I/O 管理机制，或称“I/O 主管（I/O Manager）”，这是内核管理层（Executive）的一部分。凡是跟设备驱动（包括文件系统）有关的系统调用，进入内核管理层以后就被转交给 I/O 管理，而 I/O 管理则根据调用参数将其分发到具体的设备驱动堆叠。另一方面，I/O 管理也提供着构成设备驱动框架并维持其运转的手段，例如设备驱动模块的装载和堆叠，以及模块间的逐层调用。I/O 管理的下面就是具体的设备驱动堆叠了。

在 Windows 的设备驱动框架中，下层模块向上层模块提供一个数据结构指针。但是，上层模块并不直接从这个数据结构获取具体的函数指针，更不直接使用这些函数指针调用下层模块中的函数；而是通过一些由内核提供的函数下达“I/O 请求包”即 IRP，间接地调用下层模块提供的函数，要求其执行某种操作。这就好像是向内核下一个定单，定单中告诉内核要由哪一个下层模块执行何种操作。另一方面，对于建立了形式“堆叠”的设备驱动，上层模块在运行中通常也没有如何“找到”下层模块的问题，甚至根本就不必知道其下一层是什么模块或什么设备，模块之间已在建立形式堆叠的时候固定连接好了。此时上层模块所获得的是哪一个下层模块的指针，取决于同一个堆叠中各个模块的装载次序，实际上取决于系统的配置，而相关的配置信息则最终来自相关的.inf 文件，这些信息保存在集中的数据库“注册表（Registry）”中。这样就为通过系统配置改变具体设备驱动堆叠的结构提供了更大的灵活性，主要体现在：

- 更容易在堆叠的下层实现“重定向”，即把上层模块嫁接到不同的下层模块上；
- 更容易在堆叠内部插入以“过滤驱动对象（FiDO）”为代表的“过滤模块”。

最后，设备驱动模块不是在真空中运行，需要得到内核的支持，需要由内核为其构筑起一个运行环境，这个环境的主体就是内核导出函数，此外还有一些全局的变量和数据结构。这就是 Windows 的“设备驱动开发包”DDK 中所定义（更准确地说是“声明”）的函数和变量。

事物都是在发展的，Windows 的设备驱动框架也不是一开始就这样，更不是永远这样。前面所

讲的是为实现“即插即用”所必须要有的要素，主要就是模块的动态装载以及模块堆叠的形成。有了这些要素，包括即插即用在内的分层设备驱动就可以实现了，但是当然还可以有一些附加的要求。从 Windows 98 和 Windows 2000 开始，微软定义了一种（在当时是）新的设备驱动框架，称为 WDM 即“Windows 设备驱动模型（Windows Driver Model）”。WDM 要求设备驱动模块除满足 PnP 的需要外，还必须提供两方面的功能支持：

- 对于 WMI 的支持。WMI 是“Windows 管理手段（Windows Management Instrumentation）”的缩写。WMI 与“简单网络管理规程”SNMP 相似，要求每台 Windows 主机都能应“管理器（Manager）”的要求提供包括设备驱动在内的各种状态和统计信息。这些信息从哪儿来呢？对于设备驱动，当然得要由相应的设备驱动模块提供。
- 对于电源管理的支持。有些外设能耗不小，如果有一段较长的时间没有实际使用，就没有理由不将其转入某种“省电模式”。即使是能耗不大的外设，在节能成为一个环保问题的今天，也应该在不用时使其转入省电模式。这就是电源管理要达到的目的之一。所以，微软把支持电源管理列为 WDM 的要素之一。

总之，“老式”的设备驱动（在形式上）是不分层、不堆叠的；如果形式上分层并堆叠，在微软的术语中就称为“PnP 设备驱动”。而 WDM 设备驱动，则是至少在形式上满足了上述两项附加条件的 PnP 设备驱动。对 WMI 的支持和电源管理的重要性当然不容低估，但是对于我们理解 Windows 的设备驱动框架却并非技术关键，所以后面的叙述将集中在框架的构成与实现，而忽略这两个方面。正因为这样，我们将称之为“Windows 设备驱动框架”而不是“WDM”，以免混淆。

到了 Windows Vista，技术又有了新的发展，微软又宣布了设备驱动的新格局，称为“Windows 设备驱动基础（Windows Driver Foundation）”即 WDF。WDF 把一些低速外设的驱动搬到了用户空间，从而形成了系统空间和用户空间的两个设备驱动框架。一个称为 KMDF，即“内核模式驱动框架（Kernel-Mode Driver Framework）”，另一个则称为 UMDF，即“用户模式驱动框架（User-Mode Driver Framework）”。为此，Windows Vista 在内核中增加了一种“反射器”机制，对于速度要求不高的外设，就通过反射器将来自应用软件的 I/O 请求“反射”回用户空间，由用户空间的一个 I/O 服务进程执行相应的设备驱动程序。这样，属于 UMDF 的设备驱动模块就不再是动态装入内核，而是动态装入 I/O 服务进程的用户空间。于是一方面对于设备驱动模块的技术要求降低了，即使有问题也不会引起整个系统的崩溃；另一方面也可以提高系统的安全性，因为大部分设备驱动模块将不再装入内核，这当然有利于提防恶意代码侵入内核。

如前所述，一个堆叠至少包含一个设备驱动模块，通常则有多个模块。所以单个的“老式（Legacy）”设备驱动模块同时就是一个堆叠。不过不支持结构上、形式上的堆叠并不说明不支持实质的堆叠，比方说应用软件可以打开“打印机”设备，而“打印机”设备又在其内部打开“并行口”设备，从而将经过其处理的信息通过并行口驱动进行实际的输入/输出，那当然也并无不可，只是效率比较低一些，也不太方便。在这样的情况下，“打印机”驱动模块与“并行口”驱动模块实质上就是堆叠的，同样也构成层次式的设备驱动。所以，“老式”驱动模块并非不能支持层次式的设备驱动，不能支持设备驱动的实质上的堆叠，而只是不支持软件结构形式上的堆叠。

在一个设备驱动堆叠中，最底层的模块通常是直接与外设接口（板卡、芯片）打交道的，而外

设及其接口在内核中的逻辑表现则是一些特殊的寄存器或内存单元，对这些寄存器或内存单元的访问就是对相应外设的访问。这种直接与外设打交道的低层驱动模块在 Windows 的术语中一般称为“端口（Port）”驱动。端口驱动上面的模块，则称为“中间”驱动。所谓中间，只是就其形式而言，而并不涉及模块的性质。如果从性质上分，则有“类（Class）”驱动和“过滤（Filter）”驱动。所谓类驱动当然是某一类设备的驱动，实际上是把该类设备的驱动程序的公共部分抽出来放在一个模块中，而把涉及具体设备及其物理接口的部分剥离出来放在端口驱动中。举例来说，打印机是一类设备，但是打印机有并行打印机、串行打印机，还有网络打印机；所以公共于所有这些打印机的驱动部分应该放在一个“类驱动”模块中，而类驱动模块的下面则可以根据具体的情况放上并行口驱动模块、串行口驱动模块等，这些就是端口驱动了。有时候，由于不同厂家提供的同类设备及其接口也彼此略有不同，此时厂家还会进一步把直接涉及接口操作的部分剥离出来，配套提供特定于具体设备和接口的驱动模块，称为“小端口（Miniport）”驱动，其实是“最小”、“末梢”的意思。

至于过滤驱动，则依附于类驱动或端口驱动，目的在于拦截类设备驱动与端口设备驱动之间的信息，以实现某些统计、监视、修改乃至重定向的操作。例如，假定要求对存储在磁盘上的信息进行加密，那么由端口驱动写入磁盘以及从磁盘读入的数据就都应该是已加密的，而上面的类驱动却只能按正常的算法进行处理，此时就可以在二者之间插入一个过滤驱动模块，在此模块中实现所有的加密/解密运算。

最底层的模块往往是比较复杂的，因为这种模块通常需要处理中断，还需要处理 DPC 函数即“延迟过程请求（Delayed Procedure Call）”。此外，有的底层模块还需要处理 DMA 操作（特别地，处理 DMA 操作的模块常又称为“适配器驱动（Adapter Driver）”）。而中间模块，虽然其处理的算法和流程也可能是复杂的，却并没有中断处理、DPC、DMA 这些麻烦。

有必要说明，设备驱动的堆叠是可以嵌套的。什么叫嵌套呢？就是一种设备的端口驱动可以堆叠在另一种设备的类驱动上面。以 USB 鼠标器的驱动为例，自上而下，首先当然是鼠标器的类驱动，然后是 USB 鼠标器的端口驱动。但是这个端口驱动并不直接与硬件接口打交道，因为这种鼠标器是作为 USB 设备出现的，于是在这个端口驱动模块的下面就要放上 USB 设备的类驱动，然后是具体 USB 接口芯片的端口驱动（或小端口驱动）。这样，就好像是把两个堆叠又堆叠在一起，这就是所谓嵌套。

读者也许要问，既然是 USB 鼠标器的端口驱动，为什么就不自己来实现对于 USB 芯片的驱动呢？那样当然也是可以的，但是这里有两方面的考虑。首先，那样显然不符合软件重用的原则，既然已经有了 USB 设备的类驱动和端口驱动，为什么还要再来实现一遍？再说，即使真要再来实现一遍，也难以保证其正确性。Windows 的设备驱动模块并非开源软件，虽然可以安装、使用这些模块，却不知道里面是如何实现的，所以要自己实现一遍既非易事，更不经济。其次，即使那确是易事，并且不考虑是否经济，也还是不合适。这是因为对一个具体设备或接口的操作不应“政出多门”，而应该出自同一个模块的控制，否则就得考虑如何互斥、如何串行化、如何协调的问题。在新开发的模块中或许可以把这些因素和措施考虑进去，可是对于已经存在的模块却无法再加改变。这样，势必就造成要把原来已经存在的模块也丢弃不用，而全都换成自行开发的模块，这当然是不现实的。

端口驱动本来是直接与设备打交道的，所以又称为“物理驱动”，而代表着物理驱动模块的“对

象”就称为 PDO，即“Physical Device Object”。而且，由于物理的接口总是在某个总线（如 PCI 总线、ISA 总线）上，所以又有可能被称为“总线驱动”。上层那些“中间驱动”中的“类驱动”，则又称为“功能驱动”，相应的对象则称为 FDO，即“Functional Device Object”。至于“中间驱动”中的“过滤驱动”，倒没有别的称呼，相应的对象就称为 FiDO，即“Filter Device Object”，但是按过滤驱动的安装位置在类驱动的上面或下面又有“上过滤”和“下过滤”之分。可想而知，“老式”设备驱动模块一定是 PDO。

对于 Windows 如此繁复的术语，人们常常不免为之困惑，特别是初学时真是云里雾里。微软似乎特别善于并且乐于制造许多令人吃不准摸不清的术语。

在具体的实现上，Windows 的每个设备驱动模块都由两种“对象”代表。一种是“驱动对象”，即“Driver Object”，其数据结构是 DRIVER_OBJECT；另一种是“设备对象”，即“Device Object”，其数据结构是 DEVICE_OBJECT。前者主要用来提供一组函数指针，后者则主要用来记录具体设备（在相应层次上）的各种状态，并实现形式上的堆叠（如果需要的话）。所以，DRIVER_OBJECT 基本上代表着模块所提供的各种操作，而 DEVICE_OBJECT 基本上代表着本模块所操作的对象即具体设备的有关状态信息，实际上还记录着本模块在其所在堆叠中的位置。设备对象通常是由驱动对象创建的，驱动对象在其初始化的过程中创建设备对象。同时，对于 PnP 设备驱动，驱动对象还要提供用来将其设备对象累入堆叠的 AddDevice 函数，而内核的 I/O 管理，在启动一个模块的初始化使其创建出相应的设备对象以后，就调用驱动对象所提供的 AddDevice 函数，将其设备对象累入某个堆叠。

本来，所谓“对象”的定义应该是这样：一个（或一组）数据结构以及定义于其上的操作。按这样的定义，“驱动对象”其实不成其为“对象”，而只是设备对象的一部分，只不过这里面的信息是同种设备对象所公共的部分，所以将其抽取出来，放在一个独立的数据结构中而已。正因为这样，驱动对象与设备对象之间通常是一对多的关系，实际处理的目标可以有多个，但是处理的方法却都一样。所以，驱动对象和设备对象其实是同一事物的两面，驱动对象是其中抽象的一面，而设备对象是其中具体的一面。

驱动对象与设备对象之间一对多的关系不仅体现在一个驱动对象对多个同一类型的设备对象，还可以是一个驱动对象对多个不同类型的设备对象。例如，可以开发一个 IP 网络驱动模块，其驱动对象就是 IP 驱动，里面包括了对 UDP、TCP 和 ICMP 三种 IP 规程的支持；而设备对象则可以有 UDP、TCP 和 ICMP 三种设备对象。当然，由于用的是同一个驱动对象，这些设备对象共用同一组程序入口，但是在每一个程序里面可以根据所关联的设备对象判断应该按 UDP、TCP 还是 ICMP 进行处理。所以，从实现手段的角度看这是可行的，要考虑的只是需要从程序设计的角度加以权衡，看这是否最佳的实现方案。还有，同一个驱动对象所对应的多个设备对象完全可以在不同的堆叠中出现在不同的层次上。

我们所关心的主要是 WDM 框架，而 WDM 设备驱动模块一般是堆叠的，所以我们关心设备驱动模块的堆叠。一个设备驱动堆叠中的模块完成了初始化，创建了设备对象，并将设备对象累入堆叠之后，该堆叠的结构便如图 9.1 所示：

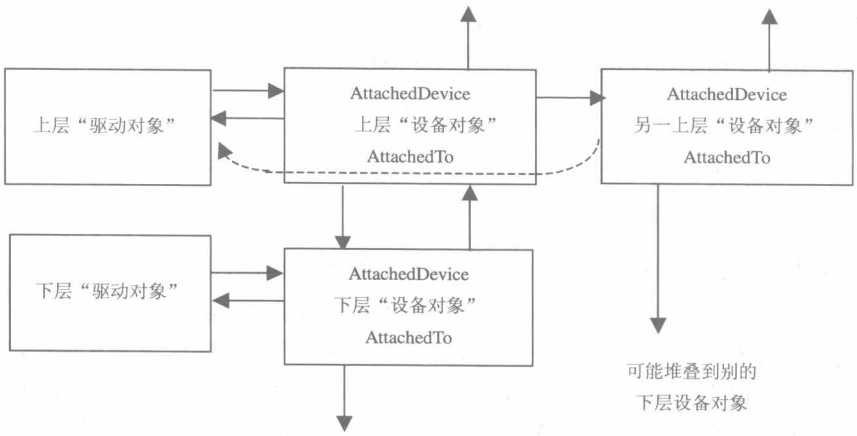

图 9.1　设备对象的堆叠和链接

这里有着两个方向上的队列。其一是横向的队列，属于同一个驱动对象的设备对象都串在一起，形成一个单链的队列，但是每个设备对象都有个指针指向其所属的驱动对象。其二是纵向的队列，这就形成设备对象的堆叠。图中的横向队列排在同一条水平线上，可能会给人一个错觉，就是这些设备对象都在同一个层次上。其实未必，同一个驱动对象的不同设备对象完全可以同时出现在（通常是不同堆叠中的）不同的层次上。即便是在同一堆叠中，实际上也并不排除在不同层次上出现同一驱动对象的不同设备对象，因为这些设备对象有着不同的参数、不同的状态信息和不同的上下层连接。

注意上下层之间连接形成堆叠的是设备对象而不是驱动对象。每个设备对象通过指针 AttachedDevice 指向其直接上层的设备对象，而上层的设备对象则通过某个指针反过来指向这个设备对象。通过什么指针呢？从原则上说，设备驱动模块的设计者需要自己在扩充部中安排一个向下的指针，所以严格说来只有各模块的设计者才知道是哪一个指针指向其下层模块的数据结构。这样，如果从最底层的模块开始，自底向上，顺着各个设备对象中的 AttachedDevice 指针上溯，是可以遍历整个堆叠的；但是反过来自顶向下就不一定了，因为在设备对象的数据结构 DEVICE_OBJECT 中并没有安排向下的指针，向下的指针只能放在设备对象的扩充部，那是由具体设备驱动模块的设计者自行设计和定义的。不过，只要设备对象扩充部的头部是 EXTENDED_DEVOBJ_EXTENSION 而不是 DEVOBJ_EXTENSION，里面就有个指针 AttachedTo，这就是向下的指针。事实上通过内核函数 IoCreateDevice()创建的设备对象都以 EXTENDED_DEVOBJ_EXTENSION 作为扩充部的头部，但是微软的资料说这一点是没有保证的，劝大家不要依靠这个指针。

所谓堆叠，虽然形式上是设备对象的堆叠，实质上却代表着驱动模块的堆叠，所以本书中根据行文的方便有时说“设备对象堆叠”，有时却又说“驱动模块堆叠”。

当上层模块要驱动（调用）下层模块，或者说要把对于设备的操作交给下一层模块去办的时候，或者 I/O 管理要调用一个模块（包括“老式”模块）时，就准备好一个称为“I/O 请求包（I/O Request Packet）”即 IRP 的数据结构，并调用一个工具性的内核函数 IoCallDriver()或 IofCallDriver()，此时程序就根据 IRP 中的“操作码”转入了目标模块所提供的某个函数。其实 IoCallDriver()是宏定义，

就定义为 IofCallDriver()，后者函数名中的‘f’是“fast”的意思，其调用界面如下：

```
FASTCALL IofCallDriver(IN PDEVICE_OBJECT DeviceObject, IN OUT PIRP Irp);
```

我们知道，快速调用函数是通过寄存器传递参数的。

这里函数名中说的是调用“驱动”而不是调用“设备”，但是实际上总是要通过具体的设备对象才能进入下一层，因为 IoCallDriver()的调用参数（之一）是设备对象指针而不是驱动对象指针。

可是上层模块或 I/O 管理机制怎么获取目标模块的设备对象指针呢？对于 I/O 管理，这是通过对象管理取得的，对象管理在打开一个设备对象时搜索对象目录，并建立起本进程与设备对象的连接（通过创建一个代表着访问上下文的文件对象，见“文件操作”一章），而连接的标识就是句柄；以后的操作则根据句柄找到目标对象的指针即可。而对于堆叠中的上层模块，则一般是根据将其累入堆叠时所建立的连接获取下层设备对象的指针。稍后读者将看到，PnP 设备对象有个扩充部分，里面有指向其下层模块设备对象的指针，而同一模块中的有关函数当然知道怎样获取和使用这个指针。在一个设备对象的数据结构中，这样的指针是否只有一个呢？换言之，上层模块与下层模块之间是否一对一的关系呢？这倒不一定。实际上一个上层模块完全可以有选择地驱动多个下层模块，就是说可以是一对多的关系，但是在典型的情况下确实只有一个。不过，反过来，如果建立了形式的堆叠，那么一个下层对象只能有一个上层对象，就是指针 AttachedDevice 所指向的设备对象。

堆叠中的上层模块可以为下层模块另行准备一个 IRP，但是一般只是把来自上层的 IRP 略加修改就往下传。

现在我们可以考察有关的数据结构了。如前所述，每个设备驱动模块（.sys 模块）都有一个 DRIVER_OBJECT 数据结构：

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;            //应该是 IO_TYPE_DRIVER
    CSHORT Size;
    // The following links all of the devices created by a single driver
    // together on a list, and the Flags word provides an extensible flag
    // location for driver objects.
    PDEVICE_OBJECT DeviceObject;      //指向第一个设备对象
    ULONG Flags;
    // The following section describes where the driver is loaded.   The count
    // field is used to count the number of times the driver has had its
    // registered reinitialization routine invoked.
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;      //指向扩充部
    // The driver name field is used by the error log thread
    // determine the name of the driver that an I/O request is/was bound.
    UNICODE_STRING DriverName;
    // The following section is for registry support.   Thise is a pointer
    // to the path to the hardware information in the registry
    PUNICODE_STRING HardwareDatabase;
    // The following section contains the optional pointer to an array of
```